Real-Time GPU-Based Motion Detection and Tracking Using Full HD Videos

Sidi Ahmed Mahmoudi[1], Michal Kierzynka[2], and Pierre Manneback[1]

[1] University of Mons, Faculty of Engineering, Mons, Belgium
[2] Poznań Supercomputing and Networking Center, Poznań, Poland

Abstract. Video processing algorithms present a necessary tool for various domains related to computer vision such as motion tracking, videos indexation and event detection. However, the new video standards, especially those in high definitions, cause that current implementations, even running on modern hardware, no longer respect the needs of real-time processing. Several solutions have been proposed to overcome this constraint, by exploiting graphic processing units (GPUs). Although, they present a high potential of GPU, any is able to treat high definition videos efficiently. In this work, we propose a development scheme enabling an efficient exploitation of GPUs, in order to achieve real-time processing of Full HD videos. Based on this scheme, we developed GPU implementations of several methods related to motion tracking such as silhouette extraction, corners detection and tracking using optical flow estimation. These implementations are exploited for improving performances of an application of real-time motion detection using mobile camera.

Keywords: GPU, CUDA, video procesing, motion tracking, real-time.

1 Introduction

In recent years, the CPU power has been capped, essentially for thermal reasons, to less than 4 GHz. A limitation that has been circumvented by the change of internal architecture, with multiplying the number of integrated computing units. This evolution is reflected in both general (CPU) and graphic (GPU) processors, as well as in recent accelerated processors (APU) which combine CPU and GPU on the same chip [1]. Moreover, GPUs have larger number of computing units, and their power has far exceeded the CPUs ones. Indeed, the advent of GPU programming interfaces (API) has encouraged many researchers to exploit them for accelerating algorithms initially designed for CPUs.

Video processing and more particularly motion estimation algorithms present the core of various methods used in computer vision. They have been used, for example, in surveillance systems tracking humans in public places, such as metro or airports, to identify possible abnormal behaviors and threats [2,3]. Motion estimation algorithms serve therefore as a common building block of some more complex routines and systems. However, these algorithms are hampered by their high consumption of both computing power and memory. The exploitation

M. Mancas et al. (Eds.): INTETAIN 2013, LNICST 124, pp. 12–21, 2013.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

of graphic processors can present an efficient solution for their acceleration. Indeed, they can present prime candidates for acceleration on GPU by exploiting its processing units in parallel, since they consist mainly of a common computation over many pixels. Nevertheless, the new standards, especially those in high resolutions cause that current implementations even running on modern hardware, no longer meet the needs of real-time processing. Moreover, modern surveillance systems are nowadays more commonly equipped with high definition cameras that expect to be treated in real-time. Furthermore, the treatment of TV broadcast images, which cannot be down sampled, require an accelerated object detection and recognition. Therfore, a fast processing of videos is needed to ensure the treatment of 25 high definition frames per second (25 fps). To overcome these constraints, several GPU computing approaches have recently been proposed. Although they present a great potential of a GPU platform, any one is able to process high definition video sequences efficiently. Thus, a need arose to develop a tool being able to address the outlined problem.

In this paper, we propose a development scheme enabling an effective exploitation of GPUs for accelerating video processing algorithms, and hence achieving real-time treatment of high definition videos. This scheme allows an efficient management of GPU memories and a fast visualization of results. Based on this scheme, we developed CUDA [4] implementations of methods related to motion tracking domain such as silhouette extraction, corners detection and tracking using optical flow estimation. These implementations are exploited for accelerating a method of real-time motion detection using mobile camera.

The remainder of the paper is organized as follows: related works are described in section 2. Section 3 presents our development scheme for video processing on GPU. Section 4 describes our GPU implementations of silhouette extraction, features detection and tracking methods. Section 5 presents the use of these implementations for impoving performance of motion detection using mobile camera. Finally, section 6 concludes and proposes further work.

2 Related Works

Unlike algorithms requiring a high dependency of computation between the input data and hence a complicated parallelization, most of image and video processing algorithms consist of similar computations over many pixels. This fact makes them well adapted for acceleration on GPU by exploiting its processing units in parallel. Otherwise, these algorithms require generally a real-time treatment of video frames. We may find several methods in this category such as human behavior understanding, event detection, camera motion estimation. These methods are generally based on motion tracking algorithms that can exploit several techniques such as optical flow estimation [6], block matching technique [7] and SIFT [8] descriptors.

Motion tracking methods consist on estimating the displacement and velocity of features in a given video frame with respect to the previous one. In this work, we are more focused on optical flow methods since they present a promising

solution for tracking even in noisy and crowded scenes or in case of small motions. In case of GPU-based optical flow motion tracking algorithms, one can find two kinds of related works. The first presents so called dense optical flow which tracks all pixels without selecting features. In this context, [9] presented a GPU implementation, using the API CUDA [4], of the Lucas-Kanade method used for optical flow estimation. The method computes dense and accurate velocity field at 15 fps with 640×480 video resolution. Authors in [11] proposed the CUDA implementation of the Horn-Schunck optical flow algorithm with a real-time processing of low resolution videos (316×252). The second category consists of methods that enable to track selected image features only. Sinha *et al.* [12] developed a GPU implementation of the KLT feature tracker [13] and the SIFT feature extraction algorithm [8]. This allowed to detect 800 features from 640×480 video at 10 fps which is around 10 times faster than the CPU implementation. However, despite their high speedups, none of the abovementioned GPU-based implementations can provide real-time processing of high definition videos. Otherwise, OpenCL [5] proposed a framework for writing programs which execute across hybrid platforms consisting of both CPUs and GPUs. There are also some GPU works dedicated to medical imaging for parallel [22] and heterogeneous [15,25] computation for vertebra detection and segmentation in X-ray images.

Our contribution focuses on the conception of a scheme development that enables an efficient exploitation of GPUs for high definition video processing in real-time. This scheme is based upon CUDA for parallel constructs and OpenGL [14] for visualization. It enables also an effective management of GPU memories that allows a fast access to pixels within video frames. Based on this scheme, we developed GPU implementations of three methods : silhouette extraction, features detection and tracking using optical flow estimation. These implementations enabled a real-time processing of Full HD videos, they were exploited for improving performance of real-time motion detection using camera in move.

3 Video Processing on GPU

As pointed out in previous sections, a GPU presents an effective tool for accelerating video processing algorithms. This section is presented in two parts: the first one describes our development scheme for video processing on GPU, showing also the employed GPU optimization techniques. The second part is devoted to describe our GPU implementations of silhouette extraction, features detection and tracking algrithms that exploit optical flow measures.

3.1 Development Scheme for Video Processing on GPU

The proposed scheme is based upon CUDA for parallel computing and OpenGL for visualization. This scheme is based on the three following steps :

1. **Loading of video frames on GPU:** we start with reading and decoding the video frames using the OpenCV library [16]. We copy the current frame on a device (GPU) that processes it in the next step.

2. **CUDA parallel processing:** before launching the parallel processing of the current frame, the number of GPU threads has to be defined, so that each thread can perform its processing on one or a group of pixels. This enables the program to treat the image pixels in parallel. Note that the number of threads depends on the number of pixels.
3. **OpenGL visualization:** the current image can be directly visualized on the screen through the video output of GPU. Therefore, we use the OpenGL library that allows for fast visualization, as it can operate buffers already existing on GPU, and thus requires less data transfer between host and device memories. Once the visualization of the current image is completed, the program goes back to the first step to load and process next frames. Otherwise and in case of multiple videos processing, the OpenGL visualization will be impossible using one video output only. So, a transfer of the processed video frames from GPU to CPU memory is required, which represents an additional cost for the application.

For a best exploitation of GPUs, we employed two optimization techniques. The first one consists on exploiting texture and shared memories. Indeed, video frames are loaded on texture memory in order to have a fast access to pixels values. The pixel neighbors are loaded on shared memory for a fast processing of pixels using their neighbors' values. The second optimization that we propose is the exploitation of four CUDA streams in order to overlap kernels executions by images transfers. Each stream consists of three instructions :

1. Copy of the current frame from host to GPU memory
2. Computations performed by CUDA kernels
3. Copy of the current frame (already processed) from GPU to host memory

3.2 GPU Implementations

Based on the scheme described in section 3.1, we propose the GPU implementation of silhouette extraction, features detection and tracking methods, which enabled to obtain both efficient results in terms of the quality of detected and tracked motions, and improved performance thanks to the exploitation of GPU.

3.2.1 GPU-Based Silhouette Extraction

The computation of difference between frames presents a simple and efficient method for detecting the silhouettes of moving objects. Based on the scheme presented in section 3.1, we propose the GPU implementation of this method using three steps. We start by loading the two first frames on GPU in order to compute the difference between them during the CUDA parallel processing step. Once the first image displayed, we replace it by the next video frame in order to apply the same treatment. Fig. 1(a) presents the obtained result of silhouette extraction. This figure shows two silhouettes extracted, that present two moving persons. In order to improve the quality of results, a threshold of 200 was used for noise elimination.

3.2.2 GPU-Based Features Detection and Tracking

In this section, we propose the GPU implementation of both features detection and tracking methods. The first one enables to detect features that are good to track, i.e. corners. To achieve this, we have exploited the Bouguet's corners extraction technique [17], which is based on the principle of Harris detector [24]. Our GPU implementation of this method is detailled in [18,19,20].

The second step enables to track the features previously detected using the optical flow method, which presents a distribution of apparent velocities of movement of brightness pattern in an image. It enables to compute the spatial displacements of images pixels based on the assumption of constant light hypothesis which supposes that the properties of consecutive images are similar in a small region. For more detail about optical flow computation, we refer readers to [6]. In literature, several optical flow methods exist such as Horn-Shunck [21], Lucas-Kanade [23] and block matching [7]. In this work, we propose the GPU implementation of the Lucas-Kanade algorithm, which is well known for its high efficiency, accuracy and robustness. This algorithm disposes of six steps:

1. **Step 1: Pyramid construction :** In the first step, the algorithm computes a pyramid representation of images I and J which represent two consecutive images from the video. The other pyramid levels are built in a recursive fashion by applying a Gaussian filter. Once the pyramid is constructed, a loop is launched that starts from the smallest image (the highest pyramid level) and ends with the original image (level 0). Its goal is to propagate the displacement vector between the pyramid levels.
2. **Step 2: Pixels matching over levels :** For each pyramid level (described in the previous step), the new coordinates of pixels (or corners) are calculated.
3. **Step 3: Local gradient computation :** In this step, the matrix of spatial gradient G is computed for each pixel (or corner) of the image I. This matrix of four elements (2×2) is calculated based on the horizontal and vertical spatial derivatives. The computation of the gradient matrix takes into account the area (window) of pixels which are centered on the point to track.
4. **Step 4: Iterative loop launch and temporal derivative computation:** A loop is launched and iterated until the difference between the two successive optical flow measures (calculated in the next step), or iterations, is higher than a defined threshold. Once the loop is launched, the computation of the temporal derivatives is performed using the image J (second image). This derivative is obtained by the subtraction of each pixel (or corner) of the image I (first image) and its corresponding corner in the image J (second image). This enables to estimate the displacement estimations which is then propagated between successive pyramid levels.
5. **Step 5: Optical flow computation:** The optical flow measure $\bar{g}$ is calculated using the gradient matrix G and the sum of temporal derivatives presented by shift vector $\bar{b}$. The measure of optical flow is calculated by multiplying the inverse of the gradient matrix G by the shift vector $\bar{b}$.
6. **Step 6: Result propagation and end of the pyramid loop:** The current results are propagated to the lower level. Once the algorithm reaches the

lowest pyramid level (the original image), the pyramid loop (launched in the first step) is stopped. The vector $\bar{g}$ presents the final optical flow value of the analyzed corner. For more detail, we refer readers to [17].

Upon matching and tracking pixels (corners) between frames, the result is a set of vectors as shown in Equation (1):

$$\Omega = \{\omega_1 \ ... \ \omega_n \mid \omega_i = (x_i, y_i, v_i, \alpha_i)\} \tag{1}$$

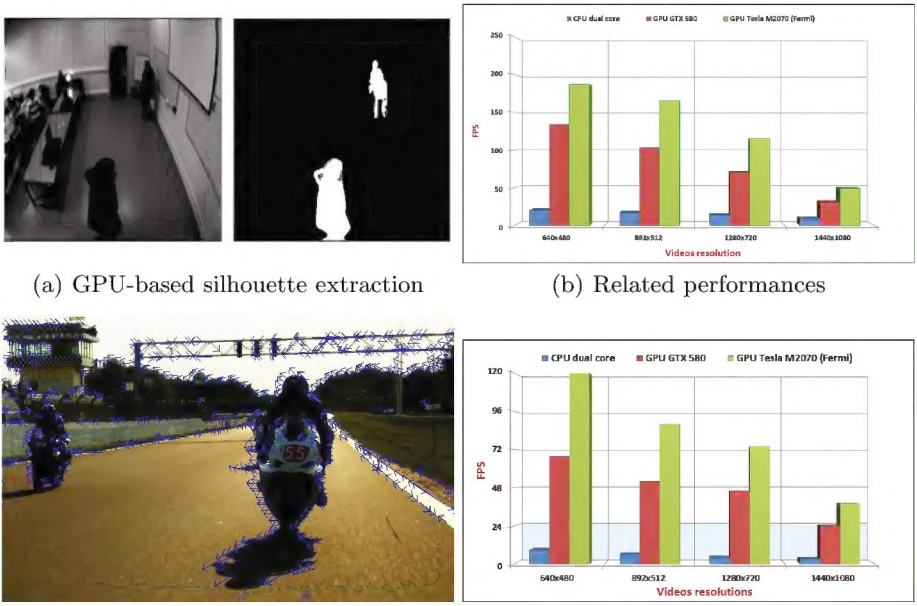
where:

- x_i, y_i are the x a y coordinates of the feature i;
- v_i represents the velocity of the feature i;
- α_i denotes motion direction of the feature i.

Based on the scheme presented in section 3.1, we propose the GPU implementation of the Lucas-Kanade otpical flow method by parallelizing its steps on GPU. These steps are executed in parallel using CUDA such that each GPU thread applies its instructions (among the six steps) on one pixel or corner. Therefore, the number of GPU threads is equal to the number of pixels or corners. Since the algorithm looks at the neighboring pixels, for a given pixel, the images, or pyramid levels are kept in the texture memory. This allows a faster access within the 2-dimensional spatial data. Other data, e.g. the arrays with computed displacements, are kept in the global memory, and are cached in the shared memory if needed. Notice that the quality of results remains identical since the process has not changed. Fig. 1(b) presents the comparison between CPU and GPU implementations of silhouette extraction method, while figures 1(c) and 1(d) present, respectively, the quality and performance of our GPU implementation of features detection and tracking method using optical flow estimation. These performances are compared with a CPU solution developed with OpenCV [16]. Notice that the constraint of real-time processing can be achieved with high definition videos thanks to the efficient exploitation of high computing power of GPUs. Notice also that the transfer time of video frames between CPU and GPU memories is included. This transfer time presents about 15 % from the total time of the application.

4 GPU for Real-Time Motion Detection Using Mobile Camera

The abovementioned GPU implementations are exploited in an application that consists of real-time motion detection within moving camera. In this category, motion detection algorithms are generally based on background subtraction which presents a widely used technique in computer vision domain. Typically, a fixed background is given to the application and new frames are subtracted from this background to detect the motion. The difference will give the objects or motion when the frame is subtracted from the fixed background. This difference in resulting binary image is called foreground objects. However, some



(a) GPU-based silhouette extraction (b) Related performances

(c) GPU-based motion tracking (d) Related performances

Fig. 1. Real-time treatment of Full HD videos on GPU

scenarios present a dynamic background which can changes due the movement of cameras. In this context, we propose an application for real-time background subtraction, which enables to detect automatically background and foreground using a moving camera. This application can be summarized in four steps :

1. **Corners detection:** The Harris corner detector [24] is applied to extract good features to track and examine for camera motion.
2. **Optical flow computation:** The Lukas-Kanade optical flow method [17] is applied to track the corners, detected previously.
3. **Camera motion Inhibition:** The camera motion is estimated by computing the dominant values of optical flow vectors. This enables to extract the common area beween each two consecutive images and focus only on motions related to objects in the scene.
4. **Motion detection:** This step consists of detecting movements based on computing the difference between each two consecutive frames.

In order to achieve a real-time treatment of high definition videos, the most intensive steps of this method are ported on GPU: corners detection, optical flow computation, motion detection. The GPU implementation of these steps is described in section 3.2, following the steps of loading of video frames on GPU, CUDA parallel processing and OpenGL visualization. Fig. 2 (a) shows a scene of camera motion. Dotted and dashed line presents the first image, dotted line presents the second frame and solid line shows the joint area of

two frames. Once, the camera motion is estimated. The joint area between 2 consecutive frames is determined by cropping the incoming and outgoing areas as seen in the white area of Fig. 2 (a). Fig. 2 (b) shows the resulting image of background subtraction. White areas represent the difference around moving objects. Table 1 presents a comparison between CPU and GPU performances of the abovementioned method. Notice that the use of GPU enabled a real-time processing for Full HD videos (1920×1080), which is 20 times faster than the corresponding CPU version.

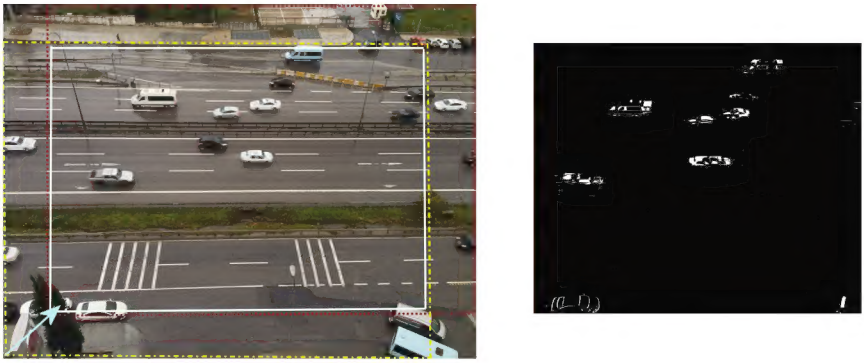


Fig. 2. (a). Camera motion estimation (b). Motion detection

Table 1. GPU performances of motion detection using mobile camera

Resolution	**CPU dual-core**	**GPU**	**Acceleration**
512×512	5 fps	79 fps	15,80 ×
1280×720	2,9 fps	51 fps	17,59 ×
1920×1080	1,7 fps	35 fps	20,59 ×

5 Conclusion

We proposed in this paper a development scheme for video processing on GPUs. Based on this scheme, we proposed an efficient implementation of the optical flow algorithm for the sparse motion tracking. More precisely, we developed a GPU based software that applies Lucas-Kanade tracking method to the previously detected corners. A GPU implementation of the silhouette extraction, based on frames difference, was also developed. These implementations were exploited for improving performance of an application that requires a real-time processing of high definition videos. This application consists of motion detection using a camera in move. As future work, we plan to develop a smart system for real-time processing of high definition videos in multi-user scenarios. This system

could exploit nvidia and ATI graphic cards thanks to the exploitation of CUDA and OpenCL APIs, respectively. The idea is to provide a dynamic platform enabling to facilitate the implementation of new advanced monitoring and control systems, effectively, that exploit parallel and heterogeneous architectures, with minimum energy consumption.

References

1. AMD Fusion, Family of APUs. The Future brought to you by AMD introducing the AMD APU Family, http://sites.amd.com/us/fusion/au/Pages/fusion.aspx
2. Fonseca, A., Mayron, L., Socek, D., Marques, O.: Design and im-plementation of an optical flow-based autonomous video surveillance system. In: Proceedings of the IASTED, p. 209 (2008)
3. Mahmoudi, S.A., Sharif, H., Ihaddadene, N., Djerabe, C.: Abnormal event detection in real time video. In: 1st International Workshop on Multimodal Interactions Analysis of Users in a Controlled Environment, ICMI (2008)
4. NVIDIA, NVIDIA CUDA: Compute Unified Device Architecture (2007), http://www.nvidia.com/cuda
5. Khronos-Group, The Open Standard for Parallel Programming of Heterogeneous Systems (2009), http://www.khronos.org/opencl
6. Bimbo, A.D., Nezi, P., Sanz, J.L.C.: Optical flow computation using extended constraints. IEEE Transaction on Image Processing, 720 (1996)
7. Kitt, B., Ranft, B., Lategahn, H.: Block-matching based optical flow estimation with reduced search space based on geometric constraints. In: 13th International Conference on Intelligent Transportation Systems, p. 1140 (2010)
8. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision (IJCV) 60(2), 91 (2004)
9. Marzat, J., Dumortier, Y., Ducrot, A.: Real-time dense and accurate parallel optical flow using CUDA. In: In Proceedings of WSCG, p. 105 (2009)
10. Mizukami, Y., Tadamura, K.: Optical Flow Computation on Compute Unified Device Architecture. In: ICIAP'14, p. 179 (2007)
11. Mizukami, Y., Tadamura, K.: Optical Flow Computation on Compute Unified Device Architecture. In: ICIAP'14, p. 179 (2007)
12. Sinha, S.N., Fram, J.-M., Pollefeys, M., Genc, Y.: Gpu-based video feature tracking and matching. In: Edge Computing Using New Commodity Architectures (2006)
13. Tomasi, C., Kanade, T.: Detection and tracking of point features. Technical Report CMU-CS-91-132, CMU, p. 1 (1991)
14. OpenGL, OpenGL Architecture Review Board: ARB vertex program, Revision 45 (2004), http://oss.sgi.com/projects/ogl-sample/registry/
15. Lecron, F., Mahmoudi, S.A., Benjelloun, M., Mahmoudi, S., Manneback, P.: Heterogeneous Computing for Vertebra Detection and Segmentation in X-Ray Images. International Journal of Biomedical Imaging (2011)
16. OpenCV, OpenCV computer vision library, http://www.opencv.org
17. Bouguet, J.Y.: Pyramidal Implementation of the Lucas Kanade Feature Tracker, Description of the algorithm. Intel Corporation Microprocessor Research (2000)
18. Mahmoudi, S.A., et al.: Traitements d'images sur architectures parallèles et hétérogènes. Technique et Science Informatiques 31, 1183 (2012)
19. Mahmoudi, S.A., Manneback, P., Augonnet, C., Thibault, S.: Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes. 20ème Rencontres Francophones du Parallélisme (2012)

20. Mahmoudi, S.A., Manneback, P.: Efficient Exploitation of Heterogeneous Platforms for Images Features Extraction. In: International Conference on Image Processing Theory, Tools and Applications, IPTA (2012)
21. Horn, B.K.P., Schunk, B.G.: Determining Optical Flow. Artificial Intelligence 2, 185 (1981)
22. Mahmoudi, S.A., Lecron, F., Manneback, P., Benjelloun, M., Mahmoudi, S.: GPU-Based Segmentation of Cervical Vertebra in X-Ray Images. In: IEEE International Conference on Cluster Computing, p. 1 (2010)
23. Lucas, B.D., Kanade, T.: An iterative image registration technique with an application to stereo vision. In: Imaging Understanding Workshop, p. 121 (1981)
24. Harris, C.: A combined corner and edge detector. In: Alvey Vision Conference, p. 147 (1988)
25. Mahmoudi, S.A., Lecron, F., Manneback, P., Benjelloun, M., Mahmoudi, S.: Efficient Exploitation of Heterogeneous Platforms for Vertebra Detection in X-Ray Images. In: Biomedical Engineering International Conference, Biomeic 2012, Tlemcen, Algeria, p. 1 (2012)